



# Training Deep Feedforwards Networks

Amir H. Payberah  
[payberah@kth.se](mailto:payberah@kth.se)  
30/11/2018



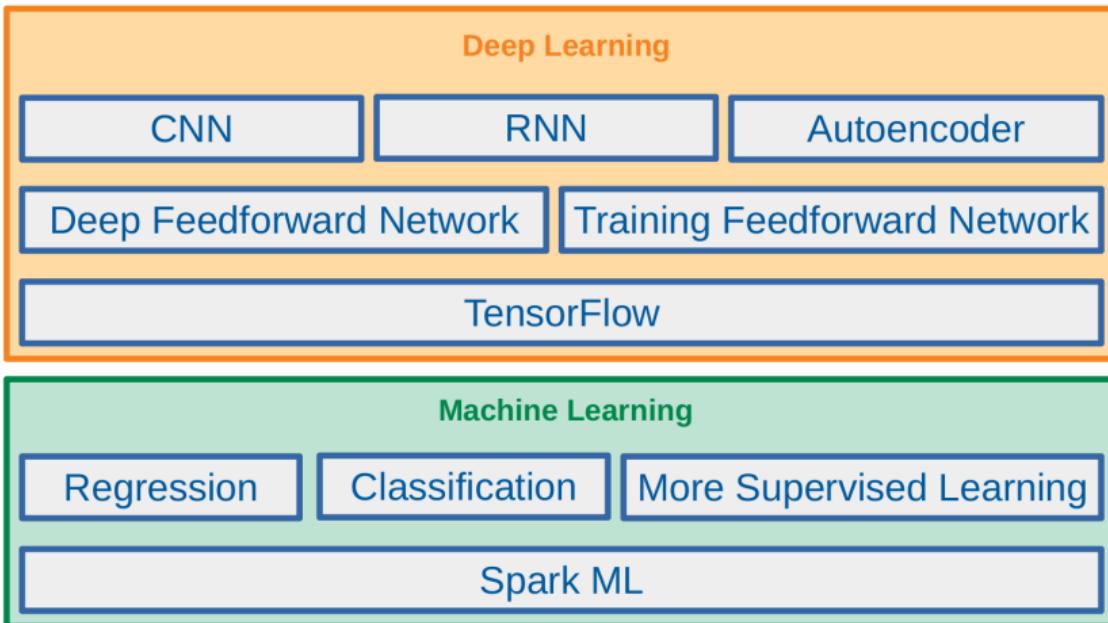


# The Course Web Page

<https://id2223kth.github.io>

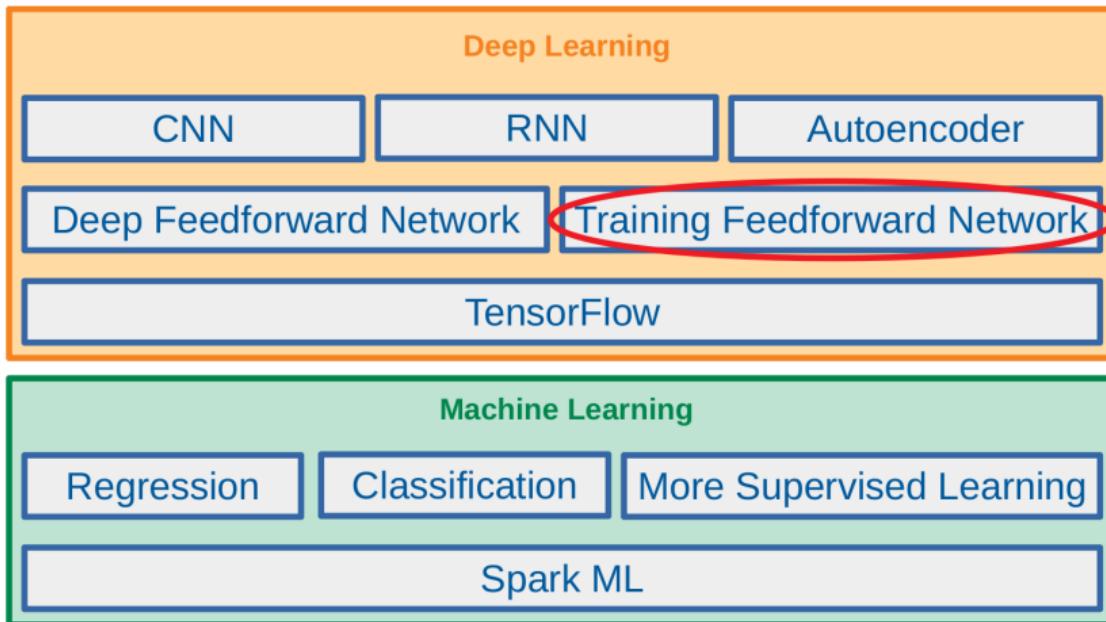


# Where Are We?





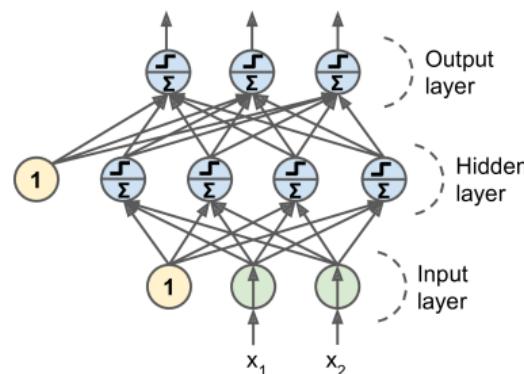
# Where Are We?



# Feedforward Neural Network Architecture

- ▶ A **feedforward neural network** is composed of:

- One **input layer**
- One or more **hidden layers**
- One final **output layer**



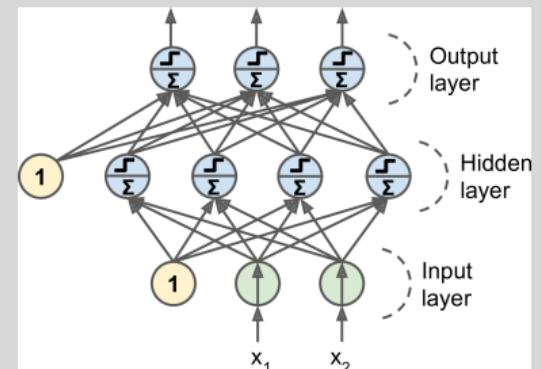
# Feedforward Network in TensorFlow (1/2)

- ▶ `n_neurons_h`: number of neurons in the hidden layer.
- ▶ `n_neurons_out`: number of neurons in the output layer.
- ▶ `n_features`: number of features.

```
n_neurons_h = 4
n_neurons_out = 3
n_features = 2

# placeholder
X = tf.placeholder(tf.float32, shape=(None, n_features),
    name="X")
y_true = tf.placeholder(tf.int64, shape=(None),
    name="y")

# make the network
hidden = tf.layers.dense(X, n_neurons_h, name="hidden",
    activation=tf.sigmoid)
logit = tf.layers.dense(hidden, n_neurons_out, name="output") # logit = Wh + b
y_hat = tf.sigmoid(logit)
```





## Feedforward Network in TensorFlow (2/2)

- ▶ Define the **cost** and the optimization, and **execute** the network.

```
# define the cost
cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(logit, y_true)
cost = tf.reduce_mean(cross_entropy)

# train the model
learning_rate = 0.1
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
training_op = optimizer.minimize(cost)

# execute the model
init = tf.global_variables_initializer()

n_epochs = 100
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        sess.run(training_op, feed_dict={X: training_X, y_true: training_y})
```



# Challenges of Training Feedforward Neural Networks

- ▶ Challenges ...





# Challenges of Training Feedforward Neural Networks

- ▶ Challenges ...
- ▶ Overfitting: risk of overfitting a model with large number of parameters.



# Challenges of Training Feedforward Neural Networks

- ▶ Challenges ...
- ▶ Overfitting: risk of overfitting a model with large number of parameters.
- ▶ Vanishing/exploding gradients: hard to train lower layers.



# Challenges of Training Feedforward Neural Networks

- ▶ Challenges ...
- ▶ Overfitting: risk of overfitting a model with large number of parameters.
- ▶ Vanishing/exploding gradients: hard to train lower layers.
- ▶ Training speed: slow training with large networks.



# Overfitting



# High Degree of Freedom and Overfitting Problem

- ▶ With **large number of parameters**, a network has a **high degree of freedom**.
- ▶ It can **fit** a huge variety of **complex datasets**.



# High Degree of Freedom and Overfitting Problem

- ▶ With **large number of parameters**, a network has a **high degree of freedom**.
- ▶ It can **fit** a huge variety of **complex datasets**.
- ▶ This **flexibility** also means that it is **prone to overfitting** on training set.



# High Degree of Freedom and Overfitting Problem

- ▶ With **large number of parameters**, a network has a **high degree of freedom**.
- ▶ It can **fit** a huge variety of **complex datasets**.
- ▶ This **flexibility** also means that it is **prone to overfitting** on training set.
- ▶ **Regularization**: a way to **reduce** the risk of **overfitting**.
- ▶ It **reduces** the **degree of freedom** a model.



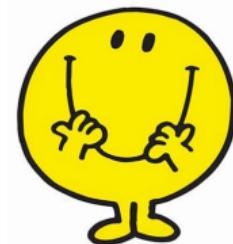
# Avoiding Overfitting Through Regularization

- ▶ Early stopping
- ▶  $\ell_1$  and  $\ell_2$  regularization
- ▶ Max-norm regularization
- ▶ Dropout
- ▶ Data augmentation



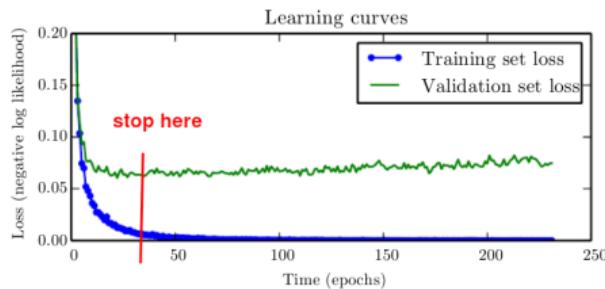
# Avoiding Overfitting Through Regularization

- ▶ Early stopping
- ▶  $\ell_1$  and  $\ell_2$  regularization
- ▶ Max-norm regularization
- ▶ Dropout
- ▶ Data augmentation



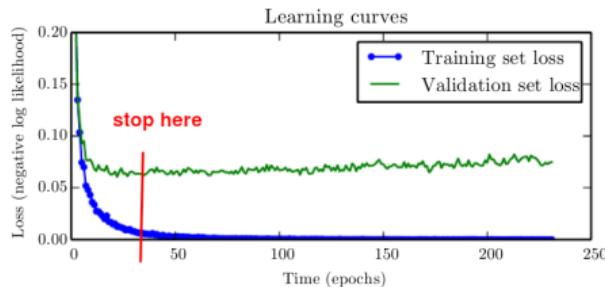
# Early Stopping

- ▶ As the **training steps go by**, its **prediction error** on the **training/validation set** naturally **goes down**.



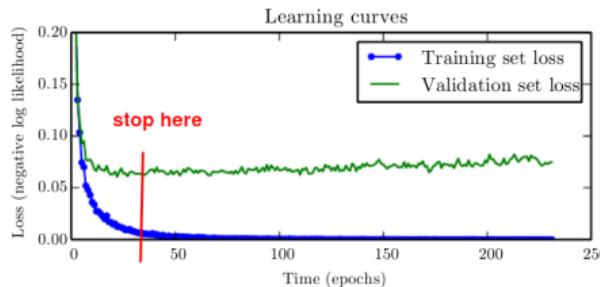
# Early Stopping

- ▶ As the **training steps go by**, its prediction error on the **training/validation set** naturally **goes down**.
- ▶ After a while the **validation error stops decreasing** and **starts to go back up**.
  - The model has started to **overfit the training data**.



# Early Stopping

- ▶ As the **training steps go by**, its **prediction error** on the **training/validation set** naturally **goes down**.
- ▶ After a while the **validation error** **stops decreasing** and **starts to go back up**.
  - The model has started to **overfit** the **training data**.
- ▶ In the **early stopping**, we **stop training** when the **validation error** reaches a **minimum**.



# Avoiding Overfitting Through Regularization

- ▶ Early stopping
- ▶  $\ell_1$  and  $\ell_2$  regularization
- ▶ Max-norm regularization
- ▶ Dropout
- ▶ Data augmentation



## $\ell_1$ and $\ell_2$ Regularization (1/4)

- ▶ Penalize **large values** of weights  $w_j$ .

$$\tilde{J}(\mathbf{w}) = J(\mathbf{w}) + \lambda R(\mathbf{w})$$

- ▶ Two questions:
  1. How should we define  $R(\mathbf{w})$ ?
  2. How do we determine  $\lambda$ ?



## /1 and /2 Regularization (2/4)

- ▶ **/1 regression:**  $R(\mathbf{w}) = \lambda \sum_{i=1}^n |w_i|$  is added to the cost function.

$$\tilde{J}(\mathbf{w}) = J(\mathbf{w}) + \lambda \sum_{i=1}^n |w_i|$$

- ▶ **/2 regression:**  $R(\mathbf{w}) = \lambda \sum_{i=1}^n w_i^2$  is added to the cost function.

$$\tilde{J}(\mathbf{w}) = J(\mathbf{w}) + \lambda \sum_{i=1}^n w_i^2$$



## $\ell_1$ and $\ell_2$ Regularization (3/4)

- ▶ Manually implement it in TensorFlow.

```
# make the network
hidden = tf.layers.dense(X, n_neurons_h, activation=tf.sigmoid, name="hidden")
logit = tf.layers.dense(hidden, n_neurons_out, name="output")
```



## $\ell_1$ and $\ell_2$ Regularization (3/4)

- ▶ Manually implement it in TensorFlow.

```
# make the network
hidden = tf.layers.dense(X, n_neurons_h, activation=tf.sigmoid, name="hidden")
logit = tf.layers.dense(hidden, n_neurons_out, name="output")

# extract the weights of layers
W1 = tf.get_default_graph().get_tensor_by_name("hidden/kernel:0")
W2 = tf.get_default_graph().get_tensor_by_name("output/kernel:0")
```



## /1 and /2 Regularization (3/4)

- ▶ Manually implement it in TensorFlow.

```
# make the network
hidden = tf.layers.dense(X, n_neurons_h, activation=tf.sigmoid, name="hidden")
logit = tf.layers.dense(hidden, n_neurons_out, name="output")

# extract the weights of layers
W1 = tf.get_default_graph().get_tensor_by_name("hidden/kernel:0")
W2 = tf.get_default_graph().get_tensor_by_name("output/kernel:0")

# l1 regularization
reg_cost = tf.reduce_sum(tf.abs(W1)) + tf.reduce_sum(tf.abs(W2))
```



## /1 and /2 Regularization (3/4)

- ▶ Manually implement it in TensorFlow.

```
# make the network
hidden = tf.layers.dense(X, n_neurons_h, activation=tf.sigmoid, name="hidden")
logit = tf.layers.dense(hidden, n_neurons_out, name="output")

# extract the weights of layers
W1 = tf.get_default_graph().get_tensor_by_name("hidden/kernel:0")
W2 = tf.get_default_graph().get_tensor_by_name("output/kernel:0")

# l1 regularization
reg_cost = tf.reduce_sum(tf.abs(W1)) + tf.reduce_sum(tf.abs(W2))

# define the cost
cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(logit, y_true)
base_cost = tf.reduce_mean(cross_entropy)

l1_param = 0.001
cost = base_cost + l1_param * reg_cost

# the rest is as before
```



## /1 and /2 Regularization (5/5)

- ▶ Alternatively, we can pass a **regularization function** to the `tf.layers.dense()`.

```
# make the network
l1_param = 0.001 # l1 regularization hyperparameter

hidden = tf.layers.dense(X, n_neurons_h, activation=tf.sigmoid, name="hidden",
    kernel_regularizer=tf.contrib.layers.l1_regularizer(l1_param))
logit = tf.layers.dense(hidden, n_neurons_out, name="output",
    kernel_regularizer=tf.contrib.layers.l1_regularizer(l1_param))
```



## /1 and /2 Regularization (5/5)

- ▶ Alternatively, we can pass a **regularization function** to the `tf.layers.dense()`.

```
# make the network
l1_param = 0.001 # l1 regularization hyperparameter

hidden = tf.layers.dense(X, n_neurons_h, activation=tf.sigmoid, name="hidden",
    kernel_regularizer=tf.contrib.layers.l1_regularizer(l1_param))
logit = tf.layers.dense(hidden, n_neurons_out, name="output",
    kernel_regularizer=tf.contrib.layers.l1_regularizer(l1_param))
```

```
# define the cost
cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(logit, y_true)

base_cost = tf.reduce_mean(cross_entropy)
reg_cost = tf.losses.get_regularization_loss()

cost = base_cost + reg_cost

# the rest is as before
```

# Avoiding Overfitting Through Regularization

- ▶ Early stopping
- ▶  $\ell_1$  and  $\ell_2$  regularization
- ▶ Max-norm regularization
- ▶ Dropout
- ▶ Data augmentation





## Max-Norm Regularization (1/3)

- ▶ Max-norm regularization: constrains the weights  $w_j$  of the incoming connections for each neuron  $j$ .
  - Prevents them from getting too large.



## Max-Norm Regularization (1/3)

- ▶ Max-norm regularization: constrains the weights  $\mathbf{w}_j$  of the incoming connections for each neuron  $j$ .
  - Prevents them from getting too large.
- ▶ After each training step, clip  $\mathbf{w}_j$  as below:

$$\mathbf{w}_j \leftarrow \mathbf{w}_j \frac{\mathbf{r}}{\|\mathbf{w}_j\|_2}$$



## Max-Norm Regularization (1/3)

- ▶ Max-norm regularization: constrains the weights  $\mathbf{w}_j$  of the incoming connections for each neuron  $j$ .
  - Prevents them from getting too large.

- ▶ After each training step, clip  $\mathbf{w}_j$  as below:

$$\mathbf{w}_j \leftarrow \mathbf{w}_j \frac{\mathbf{r}}{\|\mathbf{w}_j\|_2}$$

- ▶ We have  $\|\mathbf{w}_j\|_2 \leq \mathbf{r}$ .
  - $\mathbf{r}$  is the max-norm hyperparameter
  - $\|\mathbf{w}_j\|_2 = (\sum_i w_{i,j}^2)^{\frac{1}{2}} = \sqrt{w_{1,j}^2 + w_{2,j}^2 + \dots + w_{n,j}^2}$



## Max-Norm Regularization (2/3)

```
# make the network
hidden = tf.layers.dense(X, n_neurons_h, activation=tf.sigmoid, name="hidden")
logit = tf.layers.dense(hidden, n_neurons_out, name="output")
```



## Max-Norm Regularization (2/3)

```
# make the network
hidden = tf.layers.dense(X, n_neurons_h, activation=tf.sigmoid, name="hidden")
logit = tf.layers.dense(hidden, n_neurons_out, name="output")

# define the cost
cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(logit, y_true)
cost = tf.reduce_mean(cross_entropy)

# define the optimizer
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
training_op = optimizer.minimize(cost)
```



## Max-Norm Regularization (3/3)

- ▶ Use `tf.clip_by_norm`.

```
# max-norm regularization - hidden layer
threshold = 1.0

weights = tf.get_default_graph().get_tensor_by_name("hidden/kernel:0")
clipped_weights = tf.clip_by_norm(weights, clip_norm=threshold, axes=1)
clip_weights = weights.assign(clipped_weights)
```



## Max-Norm Regularization (3/3)

- ▶ Use `tf.clip_by_norm`.

```
# max-norm regularization - hidden layer
threshold = 1.0

weights = tf.get_default_graph().get_tensor_by_name("hidden/kernel:0")
clipped_weights = tf.clip_by_norm(weights, clip_norm=threshold, axes=1)
clip_weights = weights.assign(clipped_weights)
```

```
# executing the model
init = tf.global_variables_initializer()

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        sess.run(training_op, feed_dict={X: training_X, y_true: training_y})
        clip_weights.eval()
```

# Avoiding Overfitting Through Regularization

- ▶ Early stopping
- ▶  $\ell_1$  and  $\ell_2$  regularization
- ▶ Max-norm regularization
- ▶ Dropout
- ▶ Data augmentation



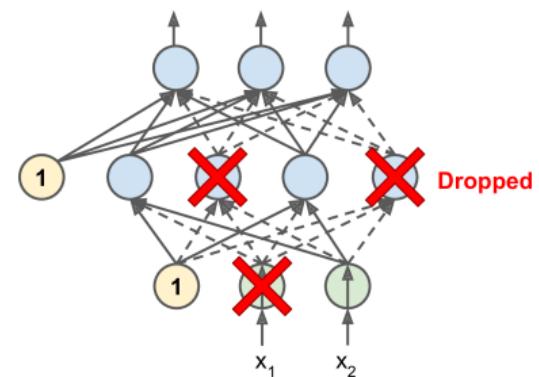
## Dropout (1/4)

- ▶ Would a **company** perform better if its employees were told **to toss a coin** every morning to decide **whether or not to go to work**?



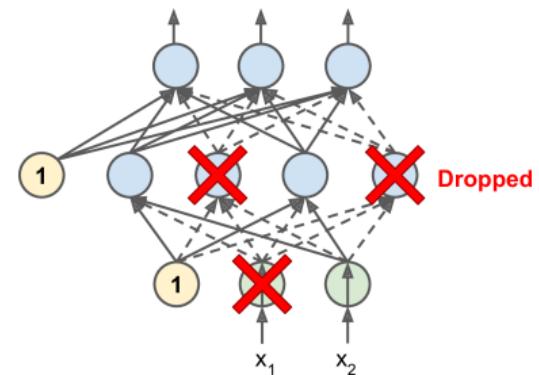
## Dropout (2/4)

- ▶ At each **training step**, each neuron drops out temporarily with a **probability  $p$** .



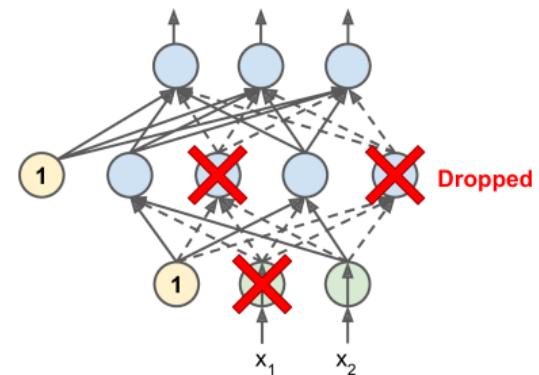
## Dropout (2/4)

- ▶ At each **training step**, each neuron drops out temporarily with a **probability  $p$** .
  - The **hyperparameter  $p$**  is called the **dropout rate**.



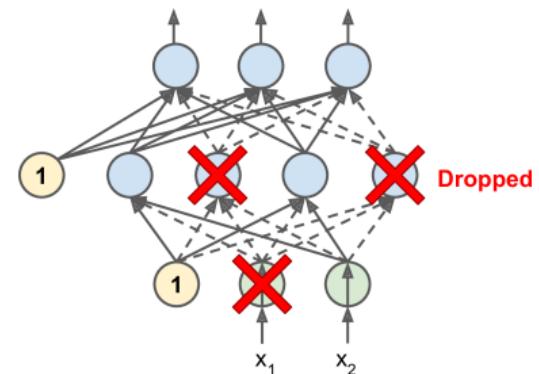
## Dropout (2/4)

- ▶ At each **training step**, each neuron drops out temporarily with a probability  $p$ .
  - The **hyperparameter  $p$**  is called the **dropout rate**.
  - A neuron will be **entirely ignored** during **this training step**.



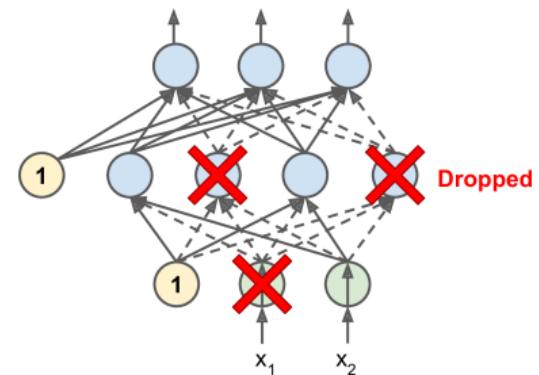
## Dropout (2/4)

- ▶ At each **training step**, each neuron drops out temporarily with a probability  $p$ .
  - The **hyperparameter  $p$**  is called the **dropout rate**.
  - A neuron will be **entirely ignored** during **this training step**.
  - It may be **active** during the **next step**.



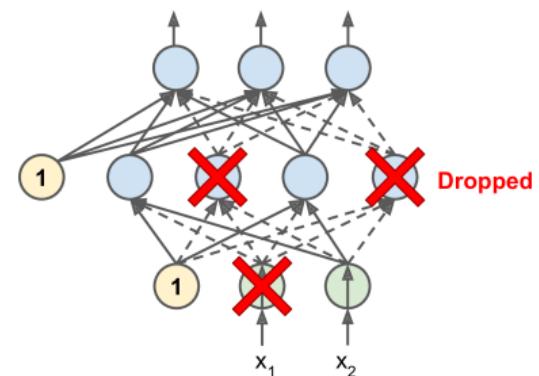
## Dropout (2/4)

- ▶ At each **training step**, each neuron drops out temporarily with a probability  $p$ .
  - The **hyperparameter  $p$**  is called the **dropout rate**.
  - A neuron will be **entirely ignored** during **this training step**.
  - It may be **active** during the **next step**.
  - Exclude the **output neurons**.



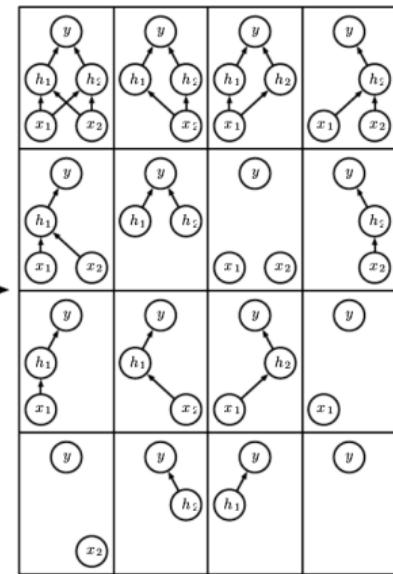
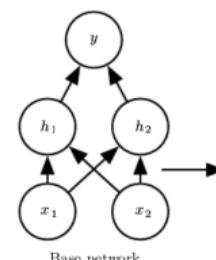
## Dropout (2/4)

- ▶ At each **training step**, each neuron drops out temporarily with a probability  $p$ .
  - The **hyperparameter  $p$**  is called the **dropout rate**.
  - A neuron will be **entirely ignored** during **this training step**.
  - It may be **active** during the **next step**.
  - Exclude the **output neurons**.
- ▶ After training, neurons don't get dropped anymore.



# Dropout (3/4)

- ▶ Each neuron can be either **present or absent**.
- ▶  **$2^N$  possible networks**, where **N** is the total number of **droppable neurons**.
  - **N = 4** in this figure.





## Dropout (4/4)

- ▶ Use `tf.layers.dropout`: specify the **dropout rate** rather than the **keep probability**.

```
# make the network
dropout_rate = 0.5 # == 1 - keep_prob
training = tf.placeholder_with_default(False, shape=(), name="training")

X_drop = tf.layers.dropout(X, dropout_rate, training=training)
hidden = tf.layers.dense(X_drop, n_neurons_h, activation=tf.sigmoid, name="hidden")
hidden_drop = tf.layers.dropout(hidden, dropout_rate, training=training)
logit = tf.layers.dense(hidden_drop, n_neurons_out, name="output")
```



## Dropout (4/4)

- ▶ Use `tf.layers.dropout`: specify the **dropout rate** rather than the **keep probability**.

```
# make the network
dropout_rate = 0.5 # == 1 - keep_prob
training = tf.placeholder_with_default(False, shape=(), name="training")

X_drop = tf.layers.dropout(X, dropout_rate, training=training)
hidden = tf.layers.dense(X_drop, n_neurons_h, activation=tf.sigmoid, name="hidden")
hidden_drop = tf.layers.dropout(hidden, dropout_rate, training=training)
logit = tf.layers.dense(hidden_drop, n_neurons_out, name="output")

# executing the model
init = tf.global_variables_initializer()

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        sess.run(training_op, feed_dict={X: training_X, y_true: training_y, training: True})
```

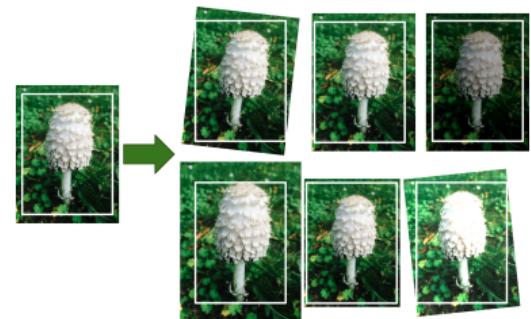
# Avoiding Overfitting Through Regularization

- ▶ Early stopping
- ▶  $\ell_1$  and  $\ell_2$  regularization
- ▶ Max-norm regularization
- ▶ Dropout
- ▶ Data augmentation



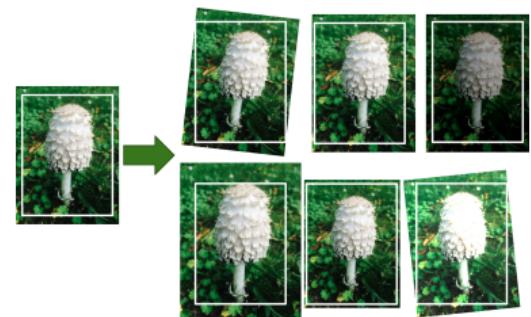
# Data Augmentation

- ▶ One way to make a model **generalize better** is to **train it on more data**.
- ▶ This will **reduce overfitting**.



# Data Augmentation

- ▶ One way to make a model **generalize better** is to **train it on more data**.
- ▶ This will **reduce overfitting**.
- ▶ Create **fake data** and add it to the **training set**.
  - E.g., in an **image classification** we can slightly shift, rotate and resize an image.
  - Add the resulting pictures to the **training set**.



# Vanishing/Exploding Gradients





## Vanishing/Exploding Gradients Problem (1/4)

- ▶ The **backpropagation** goes from **output** to **input** layer, and propagates the **error gradient** on the way.

$$\mathbf{w}^{(\text{next})} = \mathbf{w} - \eta \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$$



## Vanishing/Exploding Gradients Problem (1/4)

- ▶ The backpropagation goes from output to input layer, and propagates the error gradient on the way.

$$\mathbf{w}^{(\text{next})} = \mathbf{w} - \eta \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$$

- ▶ Gradients often get smaller and smaller as the algorithm progresses down to the lower layers.
- ▶ As a result, the gradient descent update leaves the lower layer connection weights virtually unchanged.



## Vanishing/Exploding Gradients Problem (1/4)

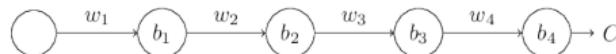
- ▶ The backpropagation goes from output to input layer, and propagates the error gradient on the way.

$$\mathbf{w}^{(\text{next})} = \mathbf{w} - \eta \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$$

- ▶ Gradients often get smaller and smaller as the algorithm progresses down to the lower layers.
- ▶ As a result, the gradient descent update leaves the lower layer connection weights virtually unchanged.
- ▶ This is called the vanishing gradients problem.

## Vanishing/Exploding Gradients Problem (2/4)

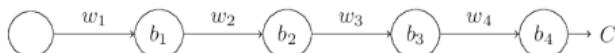
- ▶ Assume a network with just a single neuron in each layer.



- $w_1, w_2, \dots$  are the weights
- $b_1, b_2, \dots$  are the biases
- $C$  is the cost function

## Vanishing/Exploding Gradients Problem (2/4)

- ▶ Assume a network with just a single neuron in each layer.



- $w_1, w_2, \dots$  are the **weights**
- $b_1, b_2, \dots$  are the **biases**
- $C$  is the **cost function**

- ▶ The output  $a_j$  from the  $j$ th neuron is  $\sigma(z_j)$ .

- $\sigma$  is the **sigmoid** activation function
- $z_j = w_j a_{j-1} + b_j$
- E.g.,  $a_4 = \sigma(z_4) = \text{sigmoid}(w_4 a_3 + b_4)$

## Vanishing/Exploding Gradients Problem (3/4)

- Let's compute the **gradient** associated to the **first hidden neuron** ( $\frac{\partial C}{\partial b_1}$ ).



$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times \frac{\partial z_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \times \frac{\partial z_3}{\partial a_2} \times \frac{\partial a_2}{\partial z_2} \times \frac{\partial z_2}{\partial a_1} \times \frac{\partial a_1}{\partial z_1} \times \frac{\partial z_1}{\partial b_1}$$

## Vanishing/Exploding Gradients Problem (3/4)

- Let's compute the **gradient** associated to the **first hidden neuron** ( $\frac{\partial C}{\partial b_1}$ ).



$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times \frac{\partial z_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \times \frac{\partial z_3}{\partial a_2} \times \frac{\partial a_2}{\partial z_2} \times \frac{\partial z_2}{\partial a_1} \times \frac{\partial a_1}{\partial z_1} \times \frac{\partial z_1}{\partial b_1}$$

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times \frac{\partial w_4 a_3 + b_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \times \frac{\partial w_3 a_2 + b_3}{\partial a_2} \times \frac{\partial a_2}{\partial z_2} \times \frac{\partial w_2 a_1 + b_2}{\partial a_1} \times \frac{\partial a_1}{\partial z_1} \times \frac{\partial w_1 a_0 + b_1}{\partial b_1}$$

## Vanishing/Exploding Gradients Problem (3/4)

- Let's compute the **gradient** associated to the **first hidden neuron** ( $\frac{\partial C}{\partial b_1}$ ).



$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times \frac{\partial z_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \times \frac{\partial z_3}{\partial a_2} \times \frac{\partial a_2}{\partial z_2} \times \frac{\partial z_2}{\partial a_1} \times \frac{\partial a_1}{\partial z_1} \times \frac{\partial z_1}{\partial b_1}$$

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times \frac{\partial w_4 a_3 + b_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \times \frac{\partial w_3 a_2 + b_3}{\partial a_2} \times \frac{\partial a_2}{\partial z_2} \times \frac{\partial w_2 a_1 + b_2}{\partial a_1} \times \frac{\partial a_1}{\partial z_1} \times \frac{\partial w_1 a_0 + b_1}{\partial b_1}$$

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \sigma'(z_4) \times w_4 \times \sigma'(z_3) \times w_3 \times \sigma'(z_2) \times w_2 \times \sigma'(z_1) \times 1$$

## Vanishing/Exploding Gradients Problem (4/4)

- ▶ Now, consider  $\frac{\partial C}{\partial b_3}$ .



$$\frac{\partial C}{\partial b_3} = \frac{\partial C}{\partial a_4} \times \sigma'(z_4) \times w_4 \times \sigma'(z_3)$$

## Vanishing/Exploding Gradients Problem (4/4)

- ▶ Now, consider  $\frac{\partial C}{\partial b_3}$ .



$$\frac{\partial C}{\partial b_3} = \frac{\partial C}{\partial a_4} \times \sigma'(z_4) \times w_4 \times \sigma'(z_3)$$

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \sigma'(z_4) \times w_4 \times \sigma'(z_3) \times w_3 \times \sigma'(z_2) \times w_2 \times \sigma'(z_1) \times 1$$

## Vanishing/Exploding Gradients Problem (4/4)

- ▶ Now, consider  $\frac{\partial C}{\partial b_3}$ .



$$\frac{\partial C}{\partial b_3} = \frac{\partial C}{\partial a_4} \times \sigma'(z_4) \times w_4 \times \sigma'(z_3)$$

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \sigma'(z_4) \times w_4 \times \sigma'(z_3) \times w_3 \times \sigma'(z_2) \times w_2 \times \sigma'(z_1) \times 1$$

- ▶ Assume  $w_3 \sigma'(z_2) < \frac{1}{4}$  and  $w_2 \sigma'(z_1) < \frac{1}{4}$ 
  - The gradient  $\frac{\partial C}{\partial b_1}$  be a factor of 16 (or more) smaller than  $\frac{\partial C}{\partial b_3}$ .
  - This is the essential origin of the vanishing gradient problem.

# Overcoming the Vanishing Gradient

- ▶ Parameter initialization strategies
- ▶ Nonsaturating activation function
- ▶ Batch normalization
- ▶ Gradient clipping





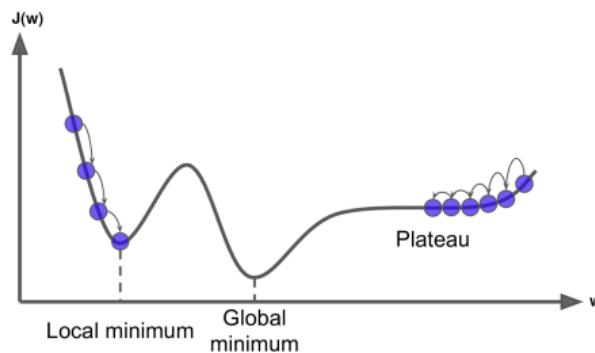
# Overcoming the Vanishing Gradient

- ▶ Parameter initialization strategies
- ▶ Nonsaturating activation function
- ▶ Batch normalization
- ▶ Gradient clipping



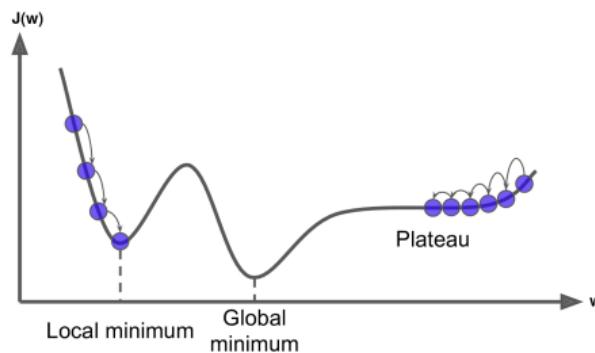
## Parameter Initialization Strategies (1/5)

- The non-linearity of a neural network causes the cost functions to become non-convex.



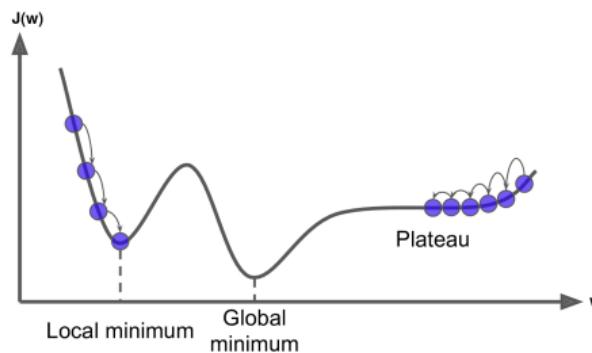
## Parameter Initialization Strategies (1/5)

- ▶ The **non-linearity** of a neural network causes the **cost functions** to become **non-convex**.
- ▶ The stochastic gradient descent on **non-convex cost functions** performs is **sensitive** to the values of the **initial parameters**.



## Parameter Initialization Strategies (1/5)

- ▶ The **non-linearity** of a neural network causes the **cost functions** to become **non-convex**.
- ▶ The stochastic gradient descent on **non-convex cost functions** performs is **sensitive** to the values of the **initial parameters**.
- ▶ Designing initialization strategies is a **difficult task**.





## Parameter Initialization Strategies (2/5)

- ▶ The **initial parameters** need to **break symmetry** between **different units**.



## Parameter Initialization Strategies (2/5)

- ▶ The **initial parameters** need to **break symmetry** between **different units**.
- ▶ **Two hidden units** with the **same activation function** connected to the **same inputs**, must have **different** initial parameters.



## Parameter Initialization Strategies (2/5)

- ▶ The **initial parameters** need to **break symmetry** between **different units**.
- ▶ **Two hidden units** with the **same activation function** connected to the **same inputs**, must have **different** initial parameters.
  - The goal of having each unit **compute a different function**.



## Parameter Initialization Strategies (2/5)

- ▶ The **initial parameters** need to **break symmetry** between **different units**.
- ▶ **Two hidden units** with the **same activation function** connected to the **same inputs**, must have **different** initial parameters.
  - The goal of having each unit **compute a different function**.
- ▶ It motivates **random initialization** of the parameters.
  - Typically, we set the **biases** to **constants**, and initialize only the **weights randomly**.



## Parameter Initialization Strategies (3/5)

- ▶ We need the signals to flow properly in **both** directions.



## Parameter Initialization Strategies (3/5)

- ▶ We need the signals to flow properly in **both** directions.
- ▶ The **Xavier initialization** proposed that:



## Parameter Initialization Strategies (3/5)

- ▶ We need the signals to flow properly in **both** directions.
- ▶ The **Xavier initialization** proposed that:
  - The **variance** of the **outputs** of each layer to be **equal** to the **variance** of its **inputs**.



## Parameter Initialization Strategies (3/5)

- ▶ We need the signals to flow properly in **both** directions.
- ▶ The **Xavier initialization** proposed that:
  - The **variance of the outputs** of each layer to be **equal** to the **variance of its inputs**.
  - The **gradients** to have **equal variance** **before and after** flowing through a layer in the reverse direction.



## Parameter Initialization Strategies (4/5)

- ▶ Based on the **Xavier initialization**, the weights are **initialized** using **normal distribution** with **mean 0** and the following **standard deviation**.



## Parameter Initialization Strategies (4/5)

- ▶ Based on the **Xavier initialization**, the weights are **initialized** using **normal distribution** with **mean 0** and the following **standard deviation**.
  - For the **sigmoid** activation function:

$$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$$

- $n_{\text{inputs}}$  and  $n_{\text{outputs}}$  are the **number of input and output connections** for the layer whose weights are being initialized.



## Parameter Initialization Strategies (4/5)

- ▶ Based on the **Xavier initialization**, the weights are **initialized** using **normal distribution** with **mean 0** and the following **standard deviation**.
  - For the **sigmoid** activation function:

$$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$$

- For the **ReLU** activation function:

$$\sigma = \sqrt{2} \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$$

- **$n_{\text{inputs}}$**  and  **$n_{\text{outputs}}$**  are the **number of input and output connections** for the layer whose weights are being initialized.



## Parameter Initialization Strategies (5/5)

- ▶ Use `tf.variance_scaling_initializer()`

```
# make the network
he_init = tf.variance_scaling_initializer()

hidden = tf.layers.dense(X, n_neurons_h, activation=tf.sigmoid, name="hidden",
    kernel_initializer=he_init)

logit = tf.layers.dense(hidden, n_neurons_out, name="output")
```

# Overcoming the Vanishing Gradient

- ▶ Parameter initialization strategies
- ▶ Nonsaturating activation function
- ▶ Batch normalization
- ▶ Gradient clipping

By Roger Hargreaves





## Nonsaturating Activation Functions (1/4)

- ▶  $\text{ReLU}(z) = \max(0, z)$
- ▶ The **dying ReLUs** problem.

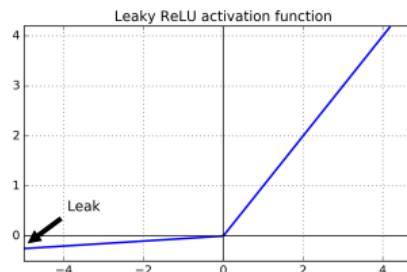


## Nonsaturating Activation Functions (1/4)

- ▶  $\text{ReLU}(z) = \max(0, z)$
- ▶ The **dying ReLUs** problem.
  - During **training**, some neurons **stop outputting anything other than 0**.
  - E.g., when the **weighted sum of the neuron's inputs is negative**, it starts outputting 0.

## Nonsaturating Activation Functions (1/4)

- ▶  $\text{ReLU}(z) = \max(0, z)$
- ▶ The **dying ReLUs** problem.
  - During **training**, some neurons **stop outputting anything other than 0**.
  - E.g., when the **weighted sum of the neuron's inputs is negative**, it starts outputting 0.
- ▶ Use **leaky ReLU** instead:  $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$ .
  - $\alpha$  is the **slope** of the function for  $z < 0$ .





## Nonsaturating Activation Functions (2/4)

### ► Randomized Leaky ReLU (RReLU)

- $\alpha$  is picked **randomly** during training, and it is **fixed** during testing.



## Nonsaturating Activation Functions (2/4)

- ▶ Randomized Leaky ReLU (RReLU)
  - $\alpha$  is picked **randomly** during training, and it is **fixed** during testing.
- ▶ Parametric Leaky ReLU (PReLU)
  - Learn  $\alpha$  **during training** (instead of being a hyperparameter).

## Nonsaturating Activation Functions (2/4)

### ► Randomized Leaky ReLU (RReLU)

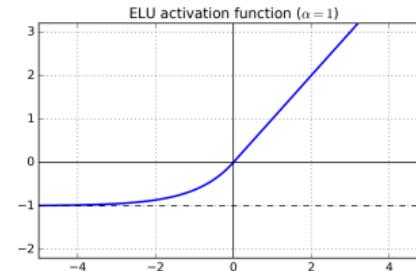
- $\alpha$  is picked **randomly** during training, and it is **fixed** during testing.

### ► Parametric Leaky ReLU (PReLU)

- Learn  $\alpha$  **during training** (instead of being a hyperparameter).

### ► Exponential Linear Unit (ELU)

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$



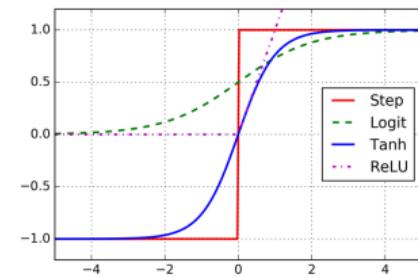
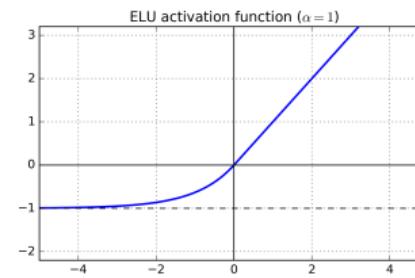
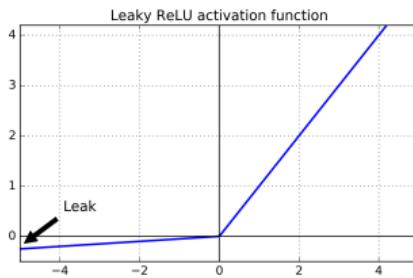


## Nonsaturating Activation Functions (3/4)

- ▶ Which activation function should we use?

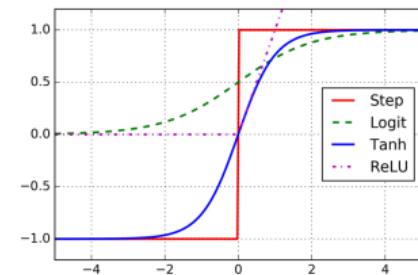
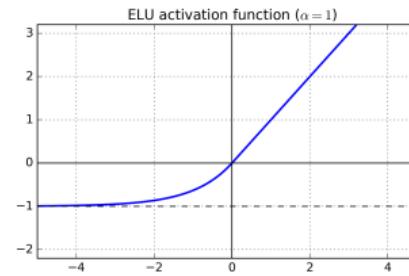
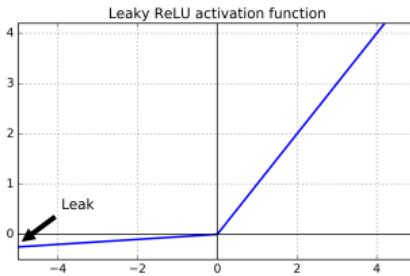
## Nonsaturating Activation Functions (3/4)

- ▶ Which activation function should we use?
- ▶ In general logistic < tanh < ReLU < leaky ReLU (and its variants) < ELU



## Nonsaturating Activation Functions (3/4)

- ▶ Which activation function should we use?
- ▶ In general logistic < tanh < ReLU < leaky ReLU (and its variants) < ELU
- ▶ If you care about runtime performance, then leaky ReLUs works better than ELUs.





## Nonsaturating Activation Functions (4/4)

```
# leaky relu
def leaky_relu(z, name=None):
    alpha = 0.01
    return tf.maximum(alpha * z, z, name=name)

hidden = tf.layers.dense(X, n_neurons_h, activation=leaky_relu, name="hidden")
```



## Nonsaturating Activation Functions (4/4)

```
# leaky relu
def leaky_relu(z, name=None):
    alpha = 0.01
    return tf.maximum(alpha * z, z, name=name)

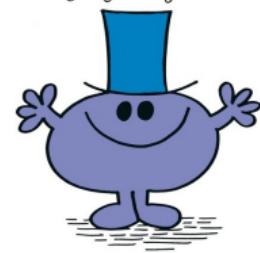
hidden = tf.layers.dense(X, n_neurons_h, activation=leaky_relu, name="hidden")
```

```
# elu
hidden = tf.layers.dense(X, n_neurons_h, activation=tf.nn.elu, name="hidden")
```

# Overcoming the Vanishing Gradient

- ▶ Parameter initialization strategies
- ▶ Nonsaturating activation function
- ▶ **Batch normalization**
- ▶ Gradient clipping

by Roger Hargreaves





## Batch Normalization (1/5)

- ▶ The gradient tells how to update each parameter, under the assumption that the other layers do not change.



## Batch Normalization (1/5)

- ▶ The gradient tells how to update each parameter, under the assumption that the other layers do not change.
  - In practice, we update all of the layers simultaneously.
  - However, unexpected results can happen.



## Batch Normalization (1/5)

- ▶ The gradient tells how to update each parameter, under the assumption that the other layers do not change.
  - In practice, we update all of the layers simultaneously.
  - However, unexpected results can happen.
- ▶ Batch normalization makes the learning of layers in the network more independent of each other.



## Batch Normalization (1/5)

- ▶ The gradient tells how to update each parameter, under the assumption that the other layers do not change.
  - In practice, we update all of the layers simultaneously.
  - However, unexpected results can happen.
- ▶ Batch normalization makes the learning of layers in the network more independent of each other.
  - It is a technique to address the problem that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change.



## Batch Normalization (1/5)

- ▶ The gradient tells how to **update each parameter**, under the assumption that **the other layers do not change**.
  - In practice, we update all of the layers **simultaneously**.
  - However, **unexpected results can happen**.
- ▶ **Batch normalization** makes the **learning of layers** in the network more **independent of each other**.
  - It is a technique to address the problem that the **distribution of each layer's inputs** changes **during training**, as the parameters of the previous layers change.
- ▶ The technique consists of **adding an operation** in the model just **before the activation function** of each layer.



## Batch Normalization (2/5)

- ▶ It's zero-centering and normalizing the inputs, then scaling and shifting the result.



## Batch Normalization (2/5)

- ▶ It's zero-centering and normalizing the inputs, then scaling and shifting the result.
  - Estimates the inputs' mean and standard deviation of the current mini-batch.

## Batch Normalization (2/5)

- ▶ It's zero-centering and normalizing the inputs, then scaling and shifting the result.
  - Estimates the inputs' mean and standard deviation of the current mini-batch.

$$\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} x^{(i)}$$

$$\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (x^{(i)} - \mu_B)^2$$

- ▶  $\mu_B$ : the empirical mean, evaluated over the whole mini-batch  $B$ .
- ▶  $\sigma_B$ : the empirical standard deviation, also evaluated over the whole mini-batch.
- ▶  $m_B$ : the number of instances in the mini-batch.



## Batch Normalization (3/5)

$$\hat{x}^{(i)} = \frac{x^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$
$$z^{(i)} = \gamma \hat{x}^{(i)} + \beta$$

- ▶  $\hat{x}^{(i)}$ : the zero-centered and normalized input.
- ▶  $\gamma$ : the scaling parameter for the layer.
- ▶  $\beta$ : the shifting parameter (offset) for the layer.
- ▶  $\epsilon$ : a tiny number to avoid division by zero.
- ▶  $z^{(i)}$ : the output of the BN operation, which is a scaled and shifted version of the inputs.



## Batch Normalization (4/5)

- ▶ Use `tf.layers.batch_normalization`

```
# make the network
training = tf.placeholder_with_default(False, shape=(), name="training")

hidden = tf.layers.dense(X, n_neurons_h, name="hidden")
bn = tf.layers.batch_normalization(hidden, training=training)
bn_act = tf.sigmoid(bn)

logits_before_bn = tf.layers.dense(bn_act, n_outputs, name="output")
logits = tf.layers.batch_normalization(logits_before_bn, training=training)
```



## Batch Normalization (4/5)

- ▶ Use `tf.layers.batch_normalization`

```
# make the network
training = tf.placeholder_with_default(False, shape=(), name="training")

hidden = tf.layers.dense(X, n_neurons_h, name="hidden")
bn = tf.layers.batch_normalization(hidden, training=training)
bn_act = tf.sigmoid(bn)

logits_before_bn = tf.layers.dense(bn_act, n_outputs, name="output")
logits = tf.layers.batch_normalization(logits_before_bn, training=training)

# define the cost
cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
cost = tf.reduce_mean(cross_entropy)

# train the model
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
training_op = optimizer.minimize(cost)
```



## Batch Normalization (5/5)

- We need to explicitly run the extra update operations needed by batch normalization  
`sess.run([training_op, extra_update_ops], ...)`

```
extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
init = tf.global_variables_initializer()

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        sess.run([training_op, extra_update_ops],
                feed_dict={X: training_X, y_true: training_y, training: True})
```

# Overcoming the Vanishing Gradient

- ▶ Parameter initialization strategies
- ▶ Nonsaturating activation function
- ▶ Batch normalization
- ▶ **Gradient clipping**

Roger Hargreaves





## Gradient Clipping (1/2)

- ▶ **Gradient clipping**: clip the gradients during **backpropagation** so that they **never exceed some threshold**.



## Gradient Clipping (1/2)

- ▶ Gradient clipping: clip the gradients during backpropagation so that they never exceed some threshold.
- ▶ In TensorFlow, the optimizer's `minimize()` function takes care of:
  1. Compute the gradients with `compute_gradients()`
  2. Apply the processed gradients with `apply_gradients()`



## Gradient Clipping (1/2)

- ▶ Gradient clipping: clip the gradients during backpropagation so that they never exceed some threshold.
- ▶ In TensorFlow, the optimizer's `minimize()` function takes care of:
  1. Compute the gradients with `compute_gradients()`
  2. Apply the processed gradients with `apply_gradients()`
- ▶ To enable the gradient clipping, you must instead of calling `minimize()`, call:
  1. Compute the gradients with `compute_gradients()`
  2. Process the gradients as you wish.
  3. Apply the processed gradients with `apply_gradients()`



## Gradient Clipping (2/2)

- ▶ Use `clip_by_value()`

```
# define the cost
cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
cost = tf.reduce_mean(cross_entropy)
```



## Gradient Clipping (2/2)

- ▶ Use `clip_by_value()`

```
# define the cost
cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
cost = tf.reduce_mean(cross_entropy)

# train the model
threshold = 1.0

optimizer = tf.train.GradientDescentOptimizer(learning_rate)

# returns a list of (gradient, variable) pairs
grads_and_vars = optimizer.compute_gradients(cost)

capped_gvs = [(tf.clip_by_value(grad, -threshold, threshold), var)
               for grad, var in grads_and_vars]

training_op = optimizer.apply_gradients(capped_gvs)
```

# Training Speed





## Regular Gradient Descent Optimization (1/2)

- ▶ Gradient descent optimization algorithm
- ▶ It updates the weights  $w_i^{(\text{next})} = w - \eta \frac{\partial J(w)}{\partial w_i}$
- ▶ Better optimization algorithms to improve the training speed



## Regular Gradient Descent Optimization (2/2)

```
# define the cost
cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(z, y_true)
cost = tf.reduce_mean(cross_entropy)

# train the model
learning_rate = 0.1
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
training_op = optimizer.minimize(cost)
```

# Optimization Algorithms

- ▶ Momentum
- ▶ Nesterov momentum
- ▶ AdaGrad
- ▶ RMSProp
- ▶ Adam Optimization



# Optimization Algorithms

- ▶ Momentum
- ▶ Nesterov momentum
- ▶ AdaGrad
- ▶ RMSProp
- ▶ Adam optimization



Roger Hargreaves

# Momentum (1/4)

- ▶ **Momentum** is a concept from physics: an **object in motion** will have a **tendency to keep moving**.
- ▶ It measures the **resistance to change in motion**.
  - The **higher momentum** an object has, the harder it is to stop it.



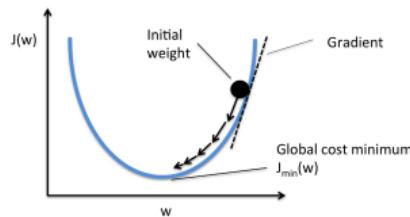


## Momentum (2/4)

- ▶ This is the very simple idea behind **momentum optimization**.

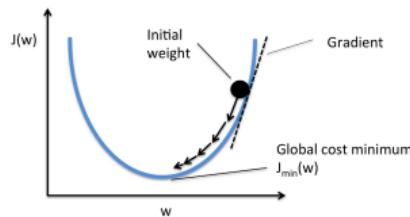
## Momentum (2/4)

- ▶ This is the very simple idea behind **momentum optimization**.
- ▶ We can see the **change in the parameters  $w$**  as **motion**:  $w_i^{(\text{next})} = w_i - \eta \frac{\partial J(w)}{\partial w_i}$



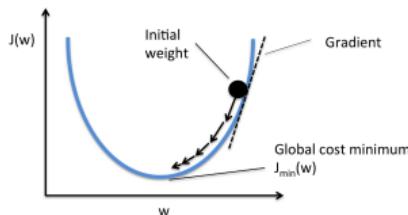
## Momentum (2/4)

- ▶ This is the very simple idea behind **momentum optimization**.
- ▶ We can see the **change in the parameters  $w$**  as **motion**:  $w_i^{(\text{next})} = w_i - \eta \frac{\partial J(w)}{\partial w_i}$
- ▶ We can thus use the concept of momentum to give the update process a **tendency to keep moving** in the same direction.



## Momentum (2/4)

- ▶ This is the very simple idea behind **momentum optimization**.
- ▶ We can see the **change in the parameters  $w$**  as **motion**:  $w_i^{(\text{next})} = w_i - \eta \frac{\partial J(w)}{\partial w_i}$
- ▶ We can thus use the concept of momentum to give the update process a **tendency to keep moving** in the same direction.
- ▶ It can help to **escape from bad local minima pits**.





## Momentum (3/4)

- ▶ Momentum optimization cares about what previous gradients were.

## Momentum (3/4)

- ▶ Momentum optimization cares about what previous gradients were.
- ▶ At each iteration, it adds the local gradient to the momentum vector  $\mathbf{m}$ .

$$\mathbf{m}_i = \beta \mathbf{m}_i + \eta \frac{\partial J(\mathbf{w})}{\partial w_i}$$

## Momentum (3/4)

- ▶ Momentum optimization cares about what previous gradients were.
- ▶ At each iteration, it adds the local gradient to the momentum vector  $\mathbf{m}$ .

$$\mathbf{m}_i = \beta \mathbf{m}_i + \eta \frac{\partial J(\mathbf{w})}{\partial w_i}$$

- ▶  $\beta$  is called momentum, and it is between 0 and 1.

## Momentum (3/4)

- ▶ Momentum optimization cares about what previous gradients were.
- ▶ At each iteration, it adds the local gradient to the momentum vector  $\mathbf{m}$ .

$$\mathbf{m}_i = \beta \mathbf{m}_i + \eta \frac{\partial J(\mathbf{w})}{\partial w_i}$$

- ▶  $\beta$  is called momentum, and it is between 0 and 1.
- ▶ Updates the weights by subtracting this momentum vector.

$$w_i^{(\text{next})} = w_i - \mathbf{m}_i$$



## Momentum (4/4)

```
# train the model  
  
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate, momentum=0.9)
```

# Optimization Algorithms

- ▶ Momentum
- ▶ Nesterov momentum
- ▶ AdaGrad
- ▶ RMSProp
- ▶ Adam optimization

By Roger Hargreaves





## Nesterov Momentum (1/3)

- ▶ Nesterov Momentum is a **small variant** to **Momentum optimization**.
- ▶ **Faster** than vanilla **Momentum optimization**.



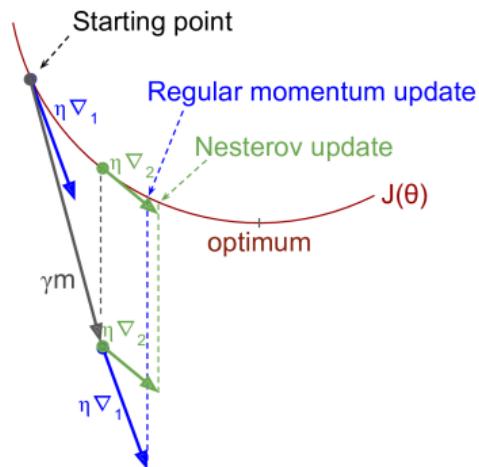
## Nesterov Momentum (1/3)

- ▶ Nesterov Momentum is a small variant to Momentum optimization.
- ▶ Faster than vanilla Momentum optimization.
- ▶ Measure the gradient of the cost function slightly ahead in the direction of the momentum (not at the local position).

$$\begin{aligned} \mathbf{m}_i &= \beta \mathbf{m}_i + \eta \frac{\partial J(\mathbf{w} + \beta \mathbf{m})}{\partial \mathbf{w}_i} \\ \mathbf{w}_i^{(\text{next})} &= \mathbf{w}_i - \mathbf{m}_i \end{aligned}$$

## Nesterov Momentum (2/3)

- ▶  $\nabla_1$  represents the **gradient of the cost function** measured at the **starting point  $w$** , and  $\nabla_2$  represents the gradient at the point located at  $w + \beta m$ .





## Nesterov Momentum (3/3)

```
# train the model

optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate, momentum=0.9,
    use_nesterov=True)
```



# Optimization Algorithms

- ▶ Momentum
- ▶ Nesterov momentum
- ▶ AdaGrad
- ▶ RMSProp
- ▶ Adam optimization





## AdaGrad (1/3)

- ▶ AdaGrad keeps track of a learning rate for each parameter.
- ▶ Adapts the learning rate over time (adaptive learning rate).



## AdaGrad (2/3)

- ▶ For each feature  $w_i$ , we do the following steps:

$$s_i = s_i + \left( \frac{\partial J(\mathbf{w})}{\partial w_i} \right)^2$$

$$w_i^{(\text{next})} = w_i - \frac{\eta}{\sqrt{s_i + \epsilon}} \frac{\partial J(\mathbf{w})}{\partial w_i}$$

- ▶ Parameters with **large** partial derivative of the cost have a **rapid decrease** in their **learning rate**.
- ▶ Parameters with **small** partial derivatives have a **small decrease** in their **learning rate**.



## AdaGrad (3/3)

```
# train the model  
  
optimizer = tf.train.AdagradOptimizer(learning_rate=learning_rate)
```

# Optimization Algorithms

- ▶ Momentum
- ▶ Nesterov momentum
- ▶ AdaGrad
- ▶ RMSProp
- ▶ Adam optimization





## RMSProp (1/3)

- ▶ AdaGrad often stops too early when training neural networks.
- ▶ The learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum.



## RMSProp (2/3)

- ▶ The **RMSProp** fixed the AdaGrad problem.
- ▶ It is like the **AdaGrad problem**, but accumulates only the gradients from the **most recent iterations** (not from the beginning of training).
- ▶ For each feature  $w_i$ , we do the following steps:

$$\begin{aligned}s_i &= \beta s_i + (1 - \beta) \left( \frac{\partial J(w)}{\partial w_i} \right)^2 \\ w_i^{(\text{next})} &= w_i - \frac{\eta}{\sqrt{s_i + \epsilon}} \frac{\partial J(w)}{\partial w_i}\end{aligned}$$



## RMSProp (3/3)

```
# train the model

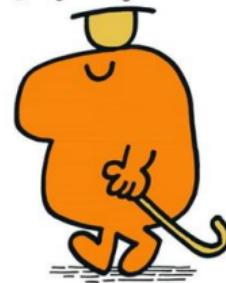
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate, momentum=0.9,
                                     decay=0.9, epsilon=1e-10)
```



# Optimization Algorithms

- ▶ Momentum
- ▶ Nesterov momentum
- ▶ AdaGrad
- ▶ RMSProp
- ▶ Adam optimization

by Roger Hargreaves





## Adam Optimization (1/3)

- ▶ Adam (Adaptive moment estimation) combines the ideas of Momentum optimization and RMSProp.



## Adam Optimization (1/3)

- ▶ Adam (Adaptive moment estimation) combines the ideas of Momentum optimization and RMSProp.
- ▶ Like Momentum optimization, it keeps track of an exponentially decaying average of past gradients.



## Adam Optimization (1/3)

- ▶ Adam (Adaptive moment estimation) combines the ideas of Momentum optimization and RMSProp.
- ▶ Like Momentum optimization, it keeps track of an exponentially decaying average of past gradients.
- ▶ Like RMSProp, it keeps track of an exponentially decaying average of past squared gradients.



## Adam Optimization (2/3)

$$1. \quad \mathbf{m}^{(\text{next})} = \beta_1 \mathbf{m} + (1 - \beta_1) \nabla_{\mathbf{w}} J(\mathbf{w})$$

$$2. \quad \mathbf{s}^{(\text{next})} = \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\mathbf{w}} J(\mathbf{w}) \otimes \nabla_{\mathbf{w}} J(\mathbf{w})$$

$$3. \quad \mathbf{m}^{(\text{next})} = \frac{\mathbf{m}}{1 - \beta_1^T}$$

$$4. \quad \mathbf{s}^{(\text{next})} = \frac{\mathbf{s}}{1 - \beta_2^T}$$

$$5. \quad \mathbf{w}^{(\text{next})} = \mathbf{w} - \eta \mathbf{m} \oslash \sqrt{\mathbf{s} + \epsilon}$$

- ▶  $\otimes$  and  $\oslash$  represents the represents the element-wise multiplication and division.
- ▶ Steps 1, 2, and 5: similar to both Momentum optimization and RMSProp.
- ▶ Steps 3 and 4: since  $\mathbf{m}$  and  $\mathbf{s}$  are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost  $\mathbf{m}$  and  $\mathbf{s}$  at the beginning of training.



## Adam Optimization (3/3)

```
# train the model  
  
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```



# Summary

# Summary

- ▶ Overfitting
  - Early stopping,  $\ell_1$  and  $\ell_2$  regularization, max-norm regularization
  - Dropout, data augmentation
- ▶ Vanishing gradient
  - Parameter initialization, nonsaturating activation functions
  - Batch normalization, gradient clipping
- ▶ Training speed
  - Momentum, nesterov momentum, AdaGrad
  - RMSProp, Adam optimization





## Reference

- ▶ Ian Goodfellow et al., Deep Learning (Ch. 7, 8)
- ▶ Aurélien Géron, Hands-On Machine Learning (Ch. 11)



# Questions?